Chapitre 1

La programmation fonctionnelle

Ce chapitre présente quelques généralités sur la programmation et introduit la programmation fonctionnelle. Sa lecture n'est pas indispensable pour comprendre les autres chapitres mais permet d'acquérir un peu de culture générale sur les langages fonctionnels et d'en comprendre « l'état d'esprit ».

1.1 Généralités sur les langages de programmation

1.1.1 Logiciel, code, langage et algorithme

Un *logiciel* est un ensemble d'actions réalisables par un ordinateur. Un navigateur web, par exemple, est un logiciel qui affiche une fenêtre à l'écran, récupère une page sur le web, l'affiche dans la fenêtre… le tout selon les demandes de l'utilisateur. Les logiciels sont très présents dans nos sociétés actuelles et peuvent prendre des formes très diverses, du navigateur web au programmateur embarqué dans la machine à laver, en passant par les robots d'analyse réseau utilisés par les moteurs de recherche.

Sans entrer dans le détail de la compilation, un logiciel est stocké sous une forme compréhensible par l'ordinateur, le *code machine*, mais qui est difficilement lisible par des humains. C'est pourquoi, les programmeurs écrivent les logiciels sous une autre forme plus compréhensible, le *code source*.

Un *langage de programmation* est l'ensemble des mots de vocabulaire et des règles de grammaire que le programmeur doit respecter pour écrire le code source d'un logiciel. Chaque langage de programmation dispose de compilateurs ou d'interpréteurs, qui permettent de traduire un code source écrit dans ce langage en code machine compréhensible par l'ordinateur.

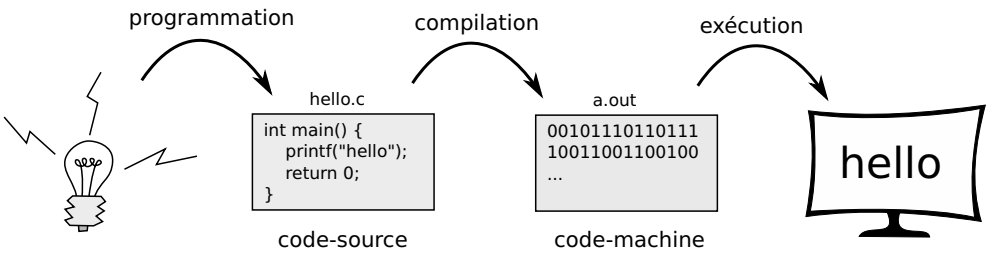

FIGURE 1.1 – Réalisation d'un logiciel : à partir d'une idée initiale ou d'un algorithme, le programmeur écrit du code source qui est ensuite compilé en code machine et peut alors être exécuté par l'ordinateur afin de réaliser l'idée initiale (ici, afficher « hello » à l'écran).

Enfin, un *algorithme* correspond à la méthode de fonctionnement suivie par un logiciel. C'est un terme très générique pour désigner les étapes à réaliser pour résoudre un problème donné. Un algorithme peut être donné sous différentes formes : code source écrit dans un langage de programmation, formule mathématique décrivant un calcul, texte en français décrivant plus ou moins précisément les étapes à réaliser… Par exemple, une formule de calcul de moyenne ou une recette de cuisine peuvent être considérées comme des algorithmes. Les étapes de la réalisation d'un logiciel, de l'algorithme à l'exécution, sont illustrées dans la figure 1.1.

1.1.2 Les paradigmes de programmation

Il existe de très nombreux langages de programmation, chacun avec ses particularités, ses avantages et ses inconvénients. Cependant, on peut généralement regrouper les langages en quelques grandes familles partageant des caractéristiques communes et un style de programmation commun, appelé *paradigme de programmation*.

La programmation impérative

Le *paradigme impératif* consiste à voir un logiciel comme une suite d'opérations de base qui changent l'état courant de l'ordinateur. Ce style de programmation est souvent bas-niveau, c'est-à-dire que les fonctionnalités supportées par le langage sont assez proches des fonctionnalités supportées directement par l'ordinateur. Par conséquent, ce paradigme est assez simple à mettre en œuvre et permet théoriquement au programmeur d'exploiter au maximum les

performances de l'ordinateur. En revanche, un code source écrit dans un langage impératif est souvent plus long car il faut décomposer l'algorithme en un nombre conséquent d'instructions basiques. De nombreux langages permettent de programmer selon le paradigme impératif, par exemple le Fortran (créé en 1954) et le C (1972).

Ci-dessous un exemple de code en langage C illustrant la programmation impérative. Dans cet exemple, l'objectif est d'avoir un compteur qui permet de compter successivement un élément puis de savoir combien d'éléments ont été comptés. Une façon possible pour implémenter cet exemple est d'utiliser une variable globale qui contient le nombre d'éléments comptés, et une fonction qui incrémente cette variable globale pour compter un élément supplémentaire. Il s'agit bien ici de programmation impérative : l'état courant de l'ordinateur (la valeur de la variable globale dans la mémoire de l'ordinateur) est modifié au fur et à mesure des instructions de code.

compteur.c

```c
#include <stdio.h>

// Variable globale stockant la valeur du compteur.
int COMPTEUR = 0;

// Fonction qui modifie la valeur de la variable globale.
void incrementerCompteur() { COMPTEUR++; }

int main() {
  incrementerCompteur();  // Modifie la variable globale.
  incrementerCompteur();
  printf("le compteur vaut %d\n", COMPTEUR); // Lit sa valeur.
  return 0;
}
```

La programmation fonctionnelle

Le *paradigme fonctionnel* consiste à voir un logiciel comme un ensemble de fonctions qui prennent des entrées et produisent des sorties, sans modifier l'état courant. C'est donc une vision plus mathématique de la programmation, diamétralement opposée au style impératif. Le style fonctionnel essaie de s'abstraire du matériel, il est donc plus difficile à mettre en œuvre, notamment de façon performante. Ainsi, les premiers langages fonctionnels (LISP, créé en 1958) sont apparus à peu près en même temps que les premiers langages impératifs mais ont mis plus de temps avant d'être réellement utilisables et utilisés.

Ci-dessous un code en Haskell illustrant comment implémenter l'exemple du compteur de la section précédente. Ici, le concept de compteur est implémenté par une valeur de compteur initiale et par une fonction qui crée un nouveau compteur à partir d'un compteur précédent (et incrémenté d'un élément). Ainsi, pour compter des éléments successifs, il suffit d'appeler la fonction d'incrémentation sur le compteur courant et de récupérer le compteur modifié, sur lequel on pourra également appeler la fonction d'incrémentation.

compteur.hs

```
1  -- Compteur initial (valeur non modifiable).
2  newCompteur = 0
3
4  -- Fonction qui retourne un nouveau compteur incrémenté.
5  incrementerCompteur compteur = compteur + 1
6
7  -- Utilisation : crée successivement un nouveau compteur à partir
8  -- d'un ancien (pas de modification de variable).
9  main = putStrLn ("le compteur vaut " ++ show cpt)
10   where cpt = incrementerCompteur (incrementerCompteur newCompteur)
```

La programmation orientée objet

Le *paradigme orienté objet* consiste à voir un logiciel comme un ensemble de briques logicielles (les objets) en interaction. Ceci permet d'organiser plus clairement les composants d'un logiciel. Par exemple, un navigateur est composé d'une fenêtre principale, d'un bouton pour revenir à la page précédente, d'un bouton d'actualisation… Chaque composant est implémenté par un objet, qui peut interagir avec d'autres objets. Enfin, certains objets peuvent correspondre à un même concept, avec des fonctionnalités communes bien définies, formant ainsi une classe d'objets. Le programmeur peut définir des classes (par exemple, pour le navigateur web, une classe bouton) avec les fonctionnalités correspondantes (afficher une icône, lancer une action en cas de clic). Il peut alors instancier des objets de ces classes pour créer véritablement les différents composants du logiciel (un bouton page précédente, un bouton actualiser…). Les langages orientés objet sont apparus plus tard que les langages impératifs mais sont aujourd'hui beaucoup utilisés. Parmi les nombreux langages supportant le paradigme orienté objet, on peut citer le Smalltalk (créé en 1972) et le Java (1995).

Ci-dessous un code en Java illustrant comment implémenter l'exemple du compteur des sections précédentes. Ici, le concept de compteur est implémenté par une classe contenant la valeur du compteur, et des fonctions membres pour construire un compteur, l'incrémenter et lire sa valeur. On peut alors ensuite

instancier un objet de cette classe de façon à créer un composant du logiciel, avec lequel on peut interagir (grâce aux fonctions de l'objet).

TestCompteur.java

```
1  // Définit une classe implémentant le concept de compteur.
2  class Compteur {
3    private int valeur;
4    public Compteur() { valeur = 0; }
5    public void incrementer() { valeur++; }
6    public int getValeur() { return valeur; }
7  }
8
9  public class TestCompteur {
10   public static void main(String[] args) {
11
12     // Crée un objet cpt, de la classe Compteur.
13     Compteur cpt = new Compteur();
14
15     // Interagit avec l'objet cpt, grâce à ses fonctions.
16     cpt.incrementer();
17     cpt.incrementer();
18     System.out.println("le compteur vaut " + cpt.getValeur());
19   }
20 }
```

Autres paradigmes

Il existe d'autres paradigmes de programmation, moins connus ou plus spécifiques. Par exemple, en *programmation logique*, le problème à traiter est décrit par un ensemble de règles de logique et de faits élémentaires, ce qui permet ensuite de calculer une solution automatiquement grâce à un moteur d'inférence. Le principal langage de programmation logique est le Prolog (1972) qui a notamment été utilisé pour des applications de planification de voyages, de traduction anglais-français et pour résoudre divers problèmes de recherche opérationnelle ou d'intelligence artificielle.

Autre exemple, la *programmation concurrente* consiste à prendre en compte le fait qu'un logiciel s'exécute en même temps que d'autres logiciels, avec lesquels il partage des ressources, ce qui apporte des problèmes supplémentaires (attentes de ressources, interblocages…). Avec le développement des réseaux d'ordinateurs et des processeurs multi-coeurs, ces problèmes sont de plus en plus d'actualité. La programmation concurrente peut être réalisée avec n'importe quel langage gérant les processus ou les threads mais certains langages prennent en compte les problèmes de concurrence dès leur conception, par exemple Erlang (1987) ou Go (2009).

À noter que de nombreux langages autorisent, au moins partiellement, plusieurs paradigmes de programmation. Par exemple, le langage Erlang a été conçu principalement pour la programmation concurrente mais est également un langage fonctionnel. Autre exemple, le langage C++ est un langage orienté objet mais c'est aussi un langage impératif, et il autorise quelques fonctionnalités de programmation fonctionnelle et concurrente.

1.1.3 Le choix d'un langage

Devant l'énorme population des langages de programmation, le choix du langage dans lequel écrire le code source d'un logiciel peut s'avérer compliqué. La plupart des langages sont *turing-complets*, c'est-à-dire équivalent à un modèle de calcul appelé machine de Turing. Cela signifie qu'un algorithme implémenté dans un langage turing-complet peut également être implémenté dans tout autre langage turing-complet. Cependant, chaque langage est généralement conçu pour des types d'applications particuliers. Ainsi, le JavaScript est couramment utilisé pour des applications web alors que le Fortran est plutôt utilisé pour du calcul numérique performant.

En pratique, le choix d'un langage de programmation est généralement dicté par l'environnement du projet : code déjà existant, bibliothèques disponibles, type d'application à développer, connaissances et expérience de l'équipe de développeurs, accès à du support ou à une communauté autour du langage… Lorsqu'il y a peu de contraintes pratiques et que le choix est vraiment ouvert, de nombreux critères peuvent être considérés : expressivité du langage, facilité à écrire du code source lisible et réutilisable, performance du code machine généré…

Enfin, indépendamment du paradigme de programmation, on remarque des tendances dans les choix de conception des langages. Ainsi, certains langages privilégient la flexibilité (typage dynamique ou faible, gestion mémoire automatique…), d'autres la sûreté (typage statique fort, immuabilité par défaut…) ou la performance (accès bas-niveau au matériel…).

1.2 Les langages fonctionnels

1.2.1 Expressions, fonctions et transparence référentielle

En programmation, une *expression* est une partie de code qui peut être évaluée et ainsi produire une valeur (par exemple, un nombre, un texte…). Une expression peut être un calcul arithmétique ou booléen, un appel de fonction, une constante… Ci-dessous un exemple en JavaScript dans l'interpréteur *Node.js* : l'expression $3^2 + 4$ est évaluée, ce qui donne la valeur 13.

```
$ node

> 3**2 + 4   // Évalue une expression arithmétique.
13
```

Une *fonction* peut être vue comme une expression pour laquelle on ne connaît pas encore la valeur de tous les termes. Les termes inconnus sont les paramètres de la fonction et sont utilisés pour définir l'expression à calculer. On peut alors ensuite évaluer la fonction en spécifiant la valeur des paramètres et obtenir le résultat du calcul complet.

```
$ node

> f1 = function(x, y) {       // Définit une fonction f1, qui
… return x**2 + y;            // prend deux paramètres x et y.
… }
[Function: f1]

> f1(3, 4)        // Évalue f1 pour les paramètres x=3 et y=4.
13
```

Une *fonction pure* retourne le résultat d'un calcul, dépendant uniquement des paramètres, et ne fait rien d'autre. Ainsi, évaluer une fonction pure produit une valeur sans modifier l'état courant de l'ordinateur. Dans l'exemple précédent, la fonction `f1` est pure mais dans l'exemple suivant, la fonction `f2` est impure car elle modifie la valeur d'une variable globale, définie en dehors de la fonction. Par conséquent, lorsqu'on appelle cette fonction, une valeur est retournée mais la variable globale est également modifiée, par *effet de bord*.

```
$ node

> r = 0                  // Définit une variable globale r.
0

> f2 = function(x, y) {
… r = x**2 + y;          // La fonction f2 retourne une valeur
… return r;              // et modifie r par effet de bord.
… }
[Function: f2]

> r                      // Valeur de r avant l'appel à f2.
0
```

```
> f2(3, 4)                // Appelle la fonction f2.
13

> r                       // Valeur de r après l'appel à f2.
13
```

Une expression satisfait la propriété de *transparence référentielle* si on obtient un programme équivalent en remplaçant cette expression par une expression de même valeur. Une fonction pure respecte la transparence référentielle.

Ainsi, dans les exemples précédents, l'expression `f1(3,4)` respecte la propriété de transparence référentielle car on obtient exactement le même programme si on remplace cette expression par sa valeur (13). En revanche, ce n'est pas le cas de l'expression `f2(3,4)` car, en plus de produire la valeur 13, la fonction `f2` modifie également la variable globale `r` (donc si on remplace l'expression par sa valeur, la variable globale n'est plus modifiée et le programme n'est plus équivalent).

La transparence référentielle interdit les effets de bord. Dans un langage pur (qui assure la propriété de transparence référentielle pour toute expression), il n'y a donc pas de donnée modifiable, d'affectation, de boucle… Dans ce contexte, on parle souvent de *données immuables* (*immutable data*) en opposition aux *données modifiables* (*mutable data*) des langages impératifs.

Enfin, une fonction (pure ou impure) peut s'appeller elle-même dans sa définition. On parle alors de *fonction récursive*. Cette notion de récursivité est essentielle en programmation fonctionnelle car elle permet d'implémenter, dans un cadre pur, des répétitions, qui sont généralement implémentées par des boucles, à effet de bord, en programmation impérative.

1.2.2 Principe de la programmation fonctionnelle

Le principe de la programmation fonctionnelle est d'imbriquer des fonctions pures. Ainsi le programme principal est lui-même une fonction qui prend des paramètres en entrée et retourne un résultat. Cette fonction est définie en terme d'autres fonctions, elles-mêmes définies par des fonctions, et ainsi de suite jusqu'à arriver à des fonctions primitives du langage [1] (voir la figure 1.2).

Ce style de programmation est donc très différent du style impératif (qui consiste à modifier l'état courant de la machine). Il est parfois appelé *programmation déclarative* car il consiste à exprimer ce que représentent les fonctions (le *quoi*) plutôt que leur fonctionnement (le *comment*).

1. John Hughes. *Why Functional Programming Matters*. Computer Journal, 32(2), 1989.